**Calhoun: The NPS Institutional Archive**

**DSpace Repository**

Theses and Dissertations | 1. Thesis and Dissertation Collection, all items

2012-09

# Masterless Distributed Computing Over Mobile Devices

Browne, James D.

Monterey, California. Naval Postgraduate School

http://hdl.handle.net/10945/17328

This publication is a work of the U.S. Government as definedin Title 17, United States Code, Section 101. As such, it is in thepublic domain, and under the provisions of Title 17, United StatesCode, Section 105, is not copyrighted in the U.S.

*Downloaded from NPS Archive: Calhoun*

http://www.nps.edu/library

Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

# NAVAL POSTGRADUATE SCHOOL

**MONTEREY, CALIFORNIA**

# THESIS

**MASTERLESS DISTRIBUTED COMPUTING OVER MOBILE DEVICES**

by

James D. Browne

September 2012

Thesis Co-Advisors: Craig Martell
Ralucca Gera

**Approved for public release; distribution is unlimited**

THIS PAGE INTENTIONALLY LEFT BLANK

| REPORT DOCUMENTATION PAGE | | | Form Approved OMB No. 0704-0188 |
|---|---|---|---|
| Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503. | | | |
| **1. AGENCY USE ONLY** ***(Leave blank)*** | **2. REPORT DATE** September 2012 | **3. REPORT TYPE AND DATES COVERED** Master's Thesis | |
| **4. TITLE AND SUBTITLE** Masterless Distributed Computing Over Mobile Devices | | | **5. FUNDING NUMBERS** |
| **6. AUTHOR(S)** James D. Browne | | | |
| **7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)** Naval Postgraduate School Monterey, CA 93943-5000 | | | **8. PERFORMING ORGANIZATION REPORT NUMBER** |
| **9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES)** N/A | | | **10. SPONSORING/MONITORING AGENCY REPORT NUMBER** |
| **11. SUPPLEMENTARY NOTES** The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol number _____N/A_____. | | | |
| **12a. DISTRIBUTION / AVAILABILITY STATEMENT** Approved for public release; distribution is unlimited | | | **12b. DISTRIBUTION CODE** A |


**13. ABSTRACT (maximum 200 words)**

It is obvious that information is becoming increasingly important in today's society. This can be seen by the widespread availability of high-speed Internet in homes and the ubiquity of smart phones. This new information centric paradigm is possible because of a large supporting infrastructure without which the Internet, the volumes of information, and the speed we can access them would not exist. The military has recognized the potential value of this trend because the greatest hindrance that any commander has is the fog of war—the absence of the information necessary to make critical decisions. On a battlefield, a commander would like to know the status and location of all of his soldiers, the same for enemy troops, and optimal strategies to accomplish their mission. Unfortunately this needed information is currently impossible to obtain in a timely manner. This thesis addresses these problems by presenting an architecture for ad-hoc distributed computing among mobile devices. Our results show that our system does indeed, as devices are added, speed up a distributed calculation and does it in a way that does not rely on the presence of a routable network. We also show that the speedup obtained nears optimal as the size of the computation necessary to calculate an update increases. Additionally, we have shown that we can chain distributed computations together resulting in a decreased amount of time needed to perform an SVD, an important step in many data-mining algorithms.


| **14. SUBJECT TERMS** Masterless, Distributed Computing, Android, Mobile Networking, Singular Value Decomposition, Truncated SVD, Rank Revealing QR Factorization | | | **15. NUMBER OF PAGES** 57 |
|---|---|---|---|
| | | | **16. PRICE CODE** |
| **17. SECURITY CLASSIFICATION OF REPORT** Unclassified | **18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified | **19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified | **20. LIMITATION OF ABSTRACT** UU |

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

**Approved for public release; distribution is unlimited**

**MASTERLESS DISTRIBUTED COMPUTING OVER MOBILE DEVICES**

James D. Browne
Major, United States Army
B.S., United States Military Academy, 2002

Submitted in partial fulfillment of the
requirements for the degree of

**MASTER OF SCIENCE IN COMPUTER SCIENCE**

**and**

**MASTER OF SCIENCE IN APPLIED MATHEMATICS**

from the

**NAVAL POSTGRADUATE SCHOOL**
**September 2012**

Author: James D. Browne

Approved by: Craig Martell
Thesis Co-Advisor

Ralucca Gera
Thesis Co-Advisor

Peter J. Denning
Chair, Department of Computer Science

Carlos Borges
Chair, Department of Applied Mathematics

THIS PAGE INTENTIONALLY LEFT BLANK

# ABSTRACT

It is obvious that information is becoming increasingly important in today's society. This can be seen by the widespread availability of high-speed Internet in homes and the ubiquity of smart phones. This new information centric paradigm is possible because of a large supporting infrastructure without which the Internet, the volumes of information, and the speed we can access them would not exist. The military has recognized the potential value of this trend because the greatest hindrance that any commander has is the fog of war—the absence of the information necessary to make critical decisions. On a battlefield, a commander would like to know the status and location of all of his soldiers, the same for enemy troops, and optimal strategies to accomplish their mission. Unfortunately this needed information is currently impossible to obtain in a timely manner. This thesis addresses these problems by presenting an architecture for ad-hoc distributed computing among mobile devices. Our results show that our system does indeed, as devices are added, speed up a distributed calculation and does it in a way that does not rely on the presence of a routable network. We also show that the speedup obtained nears optimal as the size of the computation necessary to calculate an update increases. Additionally, we have shown that we can chain distributed computations together resulting in a decreased amount of time needed to perform an SVD, an important step in many data-mining algorithms.

THIS PAGE INTENTIONALLY LEFT BLANK

# TABLE OF CONTENTS


I. INTRODUCTION....1
A. RESEARCH APPLICATION....1
B. RESEARCH QUESTION....2
C. RESULTS....3
D. ORGANIZATION OF THESIS....3

II. BACKGROUND, EXISTING TOOLS AND METHODOLOGIES....5
A. PARALLEL COMPUTING LAWS....5
1. Amdahl's Law....5
2. Gustafson's Law....6
B. ANDROID....7
1. Overview....7
2. Dalvik....7
3. Android Limitations....7
4. Research Applicability....8
C. MAPREDUCE....8
1. Overview....8
2. Execution....8
3. Handling Failures....9
4. Research Applicability....10
D. AD-HOC NETWORKS....10
E. SINGULAR VALUE DECOMPOSITION....11
1. Truncated SVD....11
2. Rank Revealing QR (RRQR) Factorization....12
3. RRQR Using Randomness....13
4. Low-Rank Matrix Approximation....14
5. Calculating the Truncated SVD....15
6. Businger Golub QR Matrix Factorization with Column Pivoting Algorithm....16

III. EXPERIMENTAL SETUP....19
A. COMPUTING ENVIRONMENT....19
1. Devices....19
2. Network....19
3. Data Storage....19
B. MASTERLESS DISTRIBUTED COMPUTATION PROCESS....20
1. Overview....20
2. Execution....20
3. Handling Failures....21
C. PRELIMINARY COMPUTATIONS....22
D. SINGULAR VALUE DECOMPOSITION COMPUTATION....22
1. Calculation of $Y=A\Omega$....22
2. Calculation of the *Q* Portion of the QR Decomposition of *Y*....23

3. Calculation of $D=Q^{T}A$ ..... 24
4. Decomposition of $D$ into its SVD ..... 24
5. Calculation of $U$ ..... 25

IV. RESULTS AND ANALYSIS ..... 27
A. OVERVIEW ..... 27
B. MATRIX MULTIPLICATION ..... 27
C. CHAIN OF TEN MATRIX MULTIPLICATIONS ..... 31
D. DISTRIBUTED SVD ..... 34

V. CONCLUSION AND FUTURE WORK ..... 37
A. SUMMARY ..... 37
B. FUTURE WORK ..... 37
1. Implement Masterless Distributed Computing System in an Ad-hoc Networking Environment ..... 37
2. Analyze the Socket Layer Receive Buffers ..... 37
3. Test Efficacy of the System when Devices are Both Entering the System and Leaving the System ..... 38
4. Analyze the System's Abilities to Work with Other Types of Computations ..... 38
C. CONCLUSION ..... 38

LIST OF REFERENCES ..... 41

INITIAL DISTRIBUTION LIST ..... 43

# LIST OF FIGURES


Figure 1. Businger and Golub QR factorization with column pivoting algorithm. From [16] ..... 17
Figure 2. Breakdown of steps involved in processing a parallelizable portion of a task. ..... 21
Figure 3. Steps needed to distribute the calculation of matrix Y. ..... 23
Figure 4. Steps needed to distribute the calculation of matrix *D*. ..... 24
Figure 5. Steps needed to distribute the calculation of matrix *U*. ..... 25
Figure 6. Speed up achieved at each matrix size. ..... 28
Figure 7. The redundancy observed during the experimental runs. ..... 29
Figure 8. Network usage during matrix multiplication. ..... 30
Figure 9. Speed up obtained by performing ten matrix multiplications. ..... 31
Figure 10. Redundancy observed in Chain of 10 Matrix Multiplications. ..... 32
Figure 11. Network usage during chain of multiplications. ..... 33
Figure 12. Difference between single and chain of multiplications. The percentage difference between the values measured during the single matrix multiplication and the chain of multiplications is clearly decreasing as the matrix size increases. ..... 34
Figure 13. Speed up of the SVD obtained on the 500x500 matrix. ..... 35
Figure 14. Average speed up of varying size SVD computations ..... 36

THIS PAGE INTENTIONALLY LEFT BLANK

# ACKNOWLEDGMENTS

I would like to thank my wife and children for making our time here in Monterey fun and memorable, our friends and family for helping take care of the kids, and my thesis advisors, Drs. Gera and Martell, for their help keeping me focused on finishing my thesis. Thank you all.

THIS PAGE INTENTIONALLY LEFT BLANK

# I. INTRODUCTION

It is obvious that information is becoming increasingly important in today's society. This can be seen by the widespread availability of high-speed Internet in homes and the ubiquity of smart phones. This new information centric paradigm is possible because of a large supporting infrastructure without which the Internet, the volumes of information, and the speed we can access them would not exist.

The military has recognized the potential value of this trend because the greatest hindrance that any commander has is the fog of war—the absence of the information necessary to make critical decisions. On a battlefield, a commander would like to know the status and location of all of his soldiers, the same for enemy troops, and optimal strategies to accomplish their mission. Unfortunately, this needed information is currently impossible to obtain in a timely manner. This thesis addresses these problems by presenting an architecture for ad-hoc distributed computing among mobile devices.

## A. RESEARCH APPLICATION

The chaos of a battlefield has many hurdles that must be overcome in order to provide information to the commander. A device that tries to provide this information must be lightweight so it does not hinder the soldier carrying it. This means that the device will necessarily have to be small like smart phones or tablets; and thus will have limited battery power and a limited processing capability. Other hurdles include the possibility of a device being destroyed and lack of an infrastructure that will allow these devices to efficiently communicate. The plus side is that data gathering tools (cameras, microphones, accelerometers, GPS, Wi-Fi, 3G, magnetometer, gyroscopes, etc.) are compact and use little power so they can easily be integrated into these devices.

Having every soldier carry a small data collecting device would result in an overflow of data that must be processed into meaningful information in order to be used. Unfortunately, this large amount of data cannot be processed because mobile devices currently do not have the necessary processing capacity. In situations like this, where large amounts of data needs to be processed, there are two techniques that are typically

used to accomplish the task; use of supercomputers and distributed computing. Both of these solutions, in their current form, have several drawbacks that prevent them from being deployed in a highly mobile battlefield environment.

Disadvantages that a supercomputer would suffer if deployed on a battlefield would be lack of a large power source needed to power the device, vulnerability to enemy attacks, and lack of mobility. An alternative to bringing a supercomputer to a battlefield would be to send the data to a processing facility, but unfortunately battlefields do not have the robust networks necessary to accomplish this. Even satellite transmissions, which require no local infrastructure, would be marred by low and very expensive bandwidth that would not be capable of sending and receiving the large amounts of data generated.

Distributed computing systems would be a much better fit for the battlefield but they too in their current state would be unable to process large amounts of data. Normally, a distributed computing environment is connected by a robust network that allows control information and results to flow quickly between nodes in the system. In a battlefield environment, this network would not exist and current ad-hoc network technology would not be able to cope with the amounts of data that can be generated [1], [2], [3]. Another problem would be master node redundancy, which is necessary in the event the master node becomes unavailable. To achieve redundancy in these systems requires a large amount of data being passed between the master node and the backup nodes, which exacerbates the network bandwidth problem stated above.

The aim of the current research is to develop a new form of distributed computing that has the following two properties:

- The system must not rely on a single master.
- The system must function in the absence of a robust network.

## B. RESEARCH QUESTION

The questions this research will answer are, “Can we construct a masterless distributed computing system that does not rely on network routing? And, if so, can the system be used for solving computationally expensive linear algebra problems, which are

commonly used to analyze large amounts of information?" We answer this question by modifying Google's well-known MapReduce system to have every device act as its own master, instead of incorporating only one master. In this way, the removal of any device, whether through loss of network connectivity or the complete destruction of the device, would not impair a computation in progress. This new system is then used to distribute the calculation of a singular value decomposition (SVD) of a large matrix, which is the building block of data mining processes such as principle component analysis, latent semantic analysis, and least squares parameter estimation.

## C. RESULTS

Our results show that our system does indeed, as devices are added, speed up a distributed calculation and does it in a way that does not rely on the presence of a routable network. We also show that the speedup obtained nears optimal as the size of the computation necessary to calculate an update increases. Additionally, we have shown that we can chain distributed computations together resulting in a decreased amount of time needed to perform an SVD, an important step in many data-mining algorithms.

## D. ORGANIZATION OF THESIS

The remainder of this thesis is organized as follows:

- Chapter II discusses prior and related work in the fields of distributed computing and provides background for the possible capabilities this system could have.
- Chapter III contains a description of the methods used to conduct the experiments.
- Chapter IV contains the results of the experiments and analysis.
- Chapter V contains the conclusion and areas for future work.

THIS PAGE INTENTIONALLY LEFT BLANK

# II. BACKGROUND, EXISTING TOOLS AND METHODOLOGIES

## A. PARALLEL COMPUTING LAWS

There are two basic laws that govern the speed up achievable by parallel computing; Amdahl's Law and Gustafson's law. These laws are used in this research to provide a theoretical baseline for the conducted experiments.

### 1. Amdahl's Law

In the late 1960s, Dr. Gene Amdahl hypothesized that advances in multiprocessor machines and parallelism must be accompanied by increases in sequential processing rates to achieve any meaningful speed increase for any given computation. His reason for this assertion was that the sequential portion of computations had remained steady at about 40% of executed instructions in the 10 years prior to his writing the landmark paper, Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. So, given $N$ processors, the expected time, $T$, to perform a given computation is

$$T = s + \frac{p}{N}$$

where $s$ is the time needed to perform the serial portion of the computation and $p$ is the time needed for the parallel portion of the computation. With this model, large scale parallel computing was thought to be prohibitively expensive because each additional machine decreased the computing time less and less [4]. And as the $\frac{p}{N}$ term in the equation approached zero, $T$ approached $s$, which was 40% of the overall time so the best speedup that you could hope to obtain was the inverse of the fraction of $s$, $\frac{1}{.40}$, or 2.5 times faster than when $N$ was 1 [4].

The system described in this thesis consists of $N$ devices that will each perform the serial and parallel portion of a given calculation so the T for a given calculation is

$$T = \frac{Ns}{N} + \frac{p}{N} = s + \frac{p}{N}$$

This shows that even though the serial portion of every calculation is performed N times the speed up achieved will be equivalent to an optimal parallel computation.

### 2. Gustafson's Law

After two decades of debate over Amdahl's Law, Dr. John Gustafson, a researcher at Sandia National Laboratories, showed that although the law $T = s + \frac{p}{N}$ was correct, one of Amdahl's premises was wrong. Amdahl assumed that the serial portion of a computation grew at the same rate that the parallel portion grew. So, if the parallel portion doubled, the serial portion would also double, thereby maintaining the 40% of executed code. Dr. Gustafson's research showed that there are many practical applications where *s* grows linearly as *p* grows exponentially, which makes large scale parallel computing highly efficient [5]. He then reformulated Amdahl's Law by setting *T* equal to 1 and making *s* a constant, which is summarized using the following formula

$$p = (1 - s)N$$

And since *p* is the time taken performing a parallel computation it can then be broken down into

$$p = \frac{d}{t}$$

where *d* is the size of the data in the parallel portion that needs to be processed and *t* is the speed at which the data can be processed. Now, by substituting for *p* and multiplying both sides by *t* gives us

$$d = (1 - s)Nt$$

the amount of data that can be processed in one time unit given *N* processors running at speed *t*. This will be used to determine the theoretical time that processing data should take. Given a baseline *d* and keeping *s* constant will provide a *t* that will then be used to determine the amount of data that can be processed for a given amount of time.

## B. ANDROID

### 1. Overview

Android is an open source software stack for mobile devices that includes an operating system, middleware and key applications. The operating system relies on Linux version 2.6 for core system services such as security, memory management, process management, network stack, and driver model. The kernel also acts as an abstraction layer between the hardware and the rest of the software stack. A set of C/C++ libraries is included and is used by various components of the Android system. These libraries are available for use by developers through the Android application framework and include a system C library that is the standard C system library tuned for embedded Linux-based devices, several 3D libraries that can take advantage of hardware 3D acceleration if available, and multiple core libraries that provides most of the functionality available in the core libraries of the Java programming language [6].

### 2. Dalvik

The Java Virtual Machine (VM), developed by Sun Microsystems, is software that executes Java bytecode. It was developed to allow the same bytecode to be run on any hardware and operating system that has implemented the Java VM, which gives Java its tag line of "write once, run anywhere." Dalvik is a VM based on the Java VM but designed with mobile devices in mind [7]. In particular, Dalvik was designed to work with limited processor speed, limited RAM, no swap space, and low power consumption. Every Android application runs in its own process, with its own instance of the Dalvik virtual machine.

### 3. Android Limitations

Android was designed to run on mobile devices that have limited processor speed and limited RAM. The developers of Android decided that the user interface (UI) was the most important aspect of the Android platform and took several precautions to ensure a good user experience. First, it is recommended that the thread controlling the UI should not perform any lengthy tasks, which prevents the screen from seemingly freezing.

Second, the allowable heap space per application is set by the device developer to between 24 and 48 megabytes; for comparison, the normal Java VM of a desktop computer allows 1 gigabyte of heap space per application.

Another big limitation of mobile devices, and thus Android, is the reliance on battery power. CPU intensive computations and large amounts of I/O will quickly drain the power from mobile devices. Dalvik was designed to optimize the bytecode to reduce power consumption but large computations can still quickly drain a battery of its power.

#### 4. Research Applicability

Android's open nature and its use of portable code make it a good choice for use in field environments but unfortunately its limitations prevent large amounts of data from being analyzed on the device. Given a robust network it would be possible to offload any CPU intensive computation to a backend server with an unlimited power source and lots of resources. This thesis provides a means of performing these large computations in the field where access to a robust network does not exist.

### C. MAPREDUCE

#### 1. Overview

MapReduce is a programming framework that distributes computations across a heterogeneous mixture of machines in a manner that hides the complexities inherent in this distribution from the programmers [8]. This abstraction is based on the map and reduce primitives present in functional programming languages. The map function is a function that is performed on each of a set of inputs and the reduce function is a function that takes the outputs of the map function and combines them into a smaller number of outputs. The output of the map function is a key/value pair that is used as the input of the reduce function.

#### 2. Execution

The inventors of the MapReduce framework describe seven steps in the execution of a MapReduce program [8].

1. The MapReduce library in the user program first splits the input files into *M* pieces of typically 16-64MB per piece (controllable by the user via an optional parameter). It then starts up many copies of the program on a cluster of machines.

2. One of the copies of the program—the master— is special. The rest are workers that are assigned work by the master. There are *M* map tasks and *R* reduce tasks to assign. The master picks idle workers and assigns each one a map task or a reduce task.

3. A worker who is assigned a map task reads the contents of the corresponding input split. It parses key/value pairs out of the input data and passes each pair to the user-defined map function. The intermediate key/value pairs produced by the map function are buffered in memory.

4. Periodically, the buffered pairs are written to local disk, partitioned into *R* regions by the partitioning function. The locations of these buffered pairs on the local disk are passed back to the master who is responsible for forwarding these locations to the reduce workers.

5. When a reduce worker is notified by the master about these locations, it uses remote procedure calls to read the buffered data from the local disks of the map workers. When a reduce worker has read all intermediate data for its partition, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together. The sorting is needed because typically many different keys map to the same reduce task. If the amount of intermediate data is too large to fit in memory, an external sort is used.

6. The reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values to the user's reduce function. The output of the reduce function is appended to a final output file for this reduce partition.

7. When all map tasks and reduce tasks have been completed, the master wakes up the user program. At this point, the MapReduce call in the user program returns back to the user code.

## 3. Handling Failures

There are two major types of failures that can happen in a MapReduce cluster: master failure and worker failure. In Google's implementation of MapReduce, there is no master node redundancy; if the master fails the program would return with an error [8]. The thought was that there was only a small chance that the master node would have problems during the execution of the process so planning for it was unnecessary.

Although a periodic checkpoint was discussed as a possible remedy to the failure of a master but it is not implemented in any of the major MapReduce distributions. Worker node failure is handled much more elegantly, the master periodically pings all worker nodes and if no response is received the work tasked to the failed node is reassigned to another machine.

#### 4. Research Applicability

The MapReduce framework has been shown to work with a wide range of data analysis algorithms using loosely coupled heterogeneous devices. This research uses this framework as a basis, using its fundamental map and reduce tenets, to produce a system that potentially works on a wide range of data analysis algorithms, uses loosely coupled heterogeneous devices, but handles failures in a fashion amenable to an unreliable network.

### D. AD-HOC NETWORKS

An ad-hoc network is a collection of devices that communicate without the use of a fixed infrastructure. Whereas the networks that we commonly use are composed of reliable links to high capacity networking equipment organized in a mesh creating redundant paths allowing fast communication between devices, ad-hoc networks rely on unreliable wireless links between devices and rely on the passing of data between these devices in an attempt to reach a given destination. Ad-hoc networks show up in a variety of settings, from military to disasters to the developing world to deep space; anywhere that fixed infrastructure is either unavailable or expensive. Computing over these networks is not trivial because network disconnections are common and persist over many time scales [9].

The theoretical bandwidth of an ad-hoc network scales with the number of nodes $n$ as

$$\frac{1}{\sqrt{nLog(n)}}$$

when assuming multiple source-destination pairs as normally seen in peer-to-peer type applications [10]. It was shown that non-local traffic patterns, in which the average source-destination distance grows with the network size, result in a rapid decrease of per node capacity [11]. So, in client-server type applications we can expect a low per node capacity unless the server is forced to remain in a centralized location, thereby minimizing the average source-destination distance within the network.

An ad-hoc network is the type of environment that the system developed in this thesis would be used in. The unreliability of these networks was the driving factor in choosing to have multiple masters to track the progress of a computation, but this reliability also precluded the use of mirrored masters. It was for these reasons that a system where each machine is its own master was developed; and since all the machines are equal peers there is no true master and hence the system is masterless.

## E. SINGULAR VALUE DECOMPOSITION

The SVD of a real matrix, $A^{m\times n}$ ( $m > n$ ), is an important matrix factorization that is commonly used for such tasks as Principal Component Analysis, Latent Semantic Analysis, and least squares parameter estimation. The factorization consists of a diagonal matrix $\Sigma^{m\times n}$, and orthogonal matrices $U^{m\times m}$ and $V^{n\times n}$ such that

$$A = U\Sigma V^T$$

where $\Sigma$ is a diagonal matrix with main diagonal consisting of the singular values $\sigma^1, \sigma^2, ..., \sigma^n$ where $\sigma^1 \geq \sigma^2 \geq ... \geq \sigma^n$.

### 1. Truncated SVD

When $A$ is rank deficient with numeric rank equal to $r$, $r \ll n \leq m$, $\Sigma$ is a diagonal matrix with

$$\sigma^1 \geq \sigma^2 \geq ... > \sigma^{r+1} \approx \sigma^{r+2} \approx ... \approx \sigma^n \approx 0$$

Because the last $n-r$ singular values of $A$ are near 0 there is no need to keep the last $m-r$ columns of $U$ or the last $n-r$ columns of $V$ that would get multiplied by these eigenvalues as we reconstruct A*, the approximation of matrix A. This gives the truncated SVD of $A$, as

$$A^{*m\times n} = U^{m\times r}\Sigma^{r\times r}\left(V^{n\times r}\right)^{T}$$

Depending on the relative sizes of $r$, $n$, and $m$; computing the truncated SVD can be much more efficient than computing the complete SVD. This truncated SVD is all that is required for most data analysis techniques since the matrix A* is the closest to A (compared to any other matrices of numeric rank r) with respect to the Frobenius norm[12]. When using highly correlated data, $r$ can be smaller than 1% of $n$, making the storage and processing of $A$ much faster with less use of space [13].

There are many techniques to compute the truncated SVD of a matrix. The algorithm used in this thesis was developed by Halko, Martinsson, and Tropp in their journal article, *Finding Structure with Randomness: Stochastic Algorithms for Constructing Approximate matrix Decompositions*, and was chosen for its reliance on matrix to matrix multiplications; an operation easily implemented in the masterless distributed computing system we propose in this thesis [14].

## 2. Rank Revealing QR (RRQR) Factorization

The QR factorization of a real matrix $A^{m\times n}\left(m>n\right)$ is the decomposition of $A$ into the product of a matrix $Q^{m\times n}$ with orthonormal columns and an upper triangular matrix $R^{n\times n}$. If

$$R=\begin{pmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{pmatrix}$$

where $R_{11}$ is an $r\times r$ matrix with minimum singular value $\sigma^{\min}$ and if $\sigma^{\min}\gg\left\|R_{22}\right\|_F$ then $A$ has numeric rank $r$ [15].

If there exists a permutation matrix $\Pi$ such that $A\Pi$ has a QR factorization $A\Pi = QR$, with $R$ in the same form as above, where $R_{11}$ is an $r \times r$ matrix with minimum singular value $\sigma^{\min}$ and $\sigma^{\min} \gg \|R_{22}\|_F$ then the factorization $A\Pi = QR$ is called an RRQR factorization of $A$.

The most common algorithm used to compute a partial QR decomposition that reveals the numeric rank of $A$ is the Businger Golub QR decomposition with column pivoting [15]. This algorithm performs successive orthogonalizations with pivoting on the columns of the matrix $A$ and halts when the Frobenius norm of the remaining columns is less than a computational tolerance, (i.e., $\approx 0$). The process results in a partial factorization

$$A^{m \times n} \approx Q^{m \times k} R^{k \times n}$$

where $Q$ is an orthonormal matrix, $R$ is a weakly upper-triangular matrix, and $k$, whether less than or greater than the numeric rank, is close to the minimal numeric rank of $A$ for which precision is achievable [12]. A weakly upper triangular matrix is a matrix where at least one permutation of its columns results in an upper triangular matrix.

### 3. RRQR Using Randomness

Instead of finding the permutation matrix, $\Pi$, that reveals the rank of $A$ it is possible instead to use a matrix $\Omega$, whose values are random and follow a Gaussian distribution with mean 0 and standard deviation 1, to form a matrix $Y$ that captures most of the activity of $A$, where $Y = A\Omega$. By activity it is meant that every column in $Y$ has a non-zero contribution from every eigendirection in $A$, so all of the range of A is represented in the resulting matrix $Y$ [15]. $\Omega$ is an $n \times (k + p)$ matrix where $k$ is the target rank of $A$ and $p$ is a small oversampling parameter. An RRQR algorithm is then performed on the matrix $Y$ to determine the rank of Y, which is easier than finding the rank of A because Y is a smaller matrix. Failure to terminate before all rows are orthonormalized means that $\Omega$ was not large enough to capture all of the activity of A, in which case columns can be added to $\Omega$, thereby adding columns to $Y$. If the steps taken

to perform the orthogonalizations were stored then it is possible to continue the RRQR process without restarting the entire algorithm from the beginning [12].

## 4. Low-Rank Matrix Approximation

A standard task in scientific computing is to determine for a given matrix *A*, an approximate factorization

$$A^{m\times n} = C^{m\times r} B^{r\times n}$$

where the inner dimension *r* is the numeric rank of *A*. Similarly to the truncated SVD, when the numeric rank is much smaller than *m* and *n*, this factorization allows *A* to be stored inexpensively, and to be multiplied to vectors or other matrices quickly. This factorization can also easily be converted to other factorizations including the SVD.

Only three steps are needed to compute the SVD of *A* from a partial factorization such as $A = CB$ [14]:

- Compute a QR factorization of *C* so that $C = QR$.
- Form the product $D = RB$, and compute the SVD of the much smaller matrix D, which results in $D = U_2 \Sigma V^T$.
- Form the product $U = QU_2$.

This results in the following chain of equations leading to the SVD of *A*

$$A = CB$$

Performing a QR factorization on *C* gives

$$A = QRB$$

Multiplying the matrices *R* and *B* together gives

$$A = QD$$

Now finding the SVD of *D* gives

$$A = QU_2 \Sigma V^T$$

And, finally, multiplying *Q* and $U_2$ gives us the final SVD of *A* as

$$A = U\Sigma V^T$$

Notice, if $C$ is an $m \times r$ matrix and $B$ is an $r \times n$ matrix the resulting SVD has the following dimensions

$$A^{m\times n} = U^{m\times r}\Sigma^{r\times r}\left(V^{n\times r}\right)^T$$

This is a truncated SVD.

## 5. Calculating the Truncated SVD

The first step in calculating the truncated SVD is to compute an approximate basis for the range of the input matrix $A$. In other words, we require a matrix $Q$ that has orthonormal columns and $A \approx QQ^T A$. We would like the basis matrix $Q$ to contain as few columns as possible, but it is even more important to have an accurate approximation of the input matrix. The randomized algorithm for this task is to draw a random matrix $\Omega^{n\times(k+p)}$, where $k$ is the approximate numeric rank of $A$ and $p$ is a small ($p$<10) oversampling parameter, and form the matrix product $Y = A\Omega$. Then using an RRQR algorithm on $Y$, we compute an orthonormal matrix $Q^{m\times r}$ that should satisfy $\left\|A - QQ^T A\right\|_F \approx e$, where $e$ is machine error.

Using the computed $Q$ we can now easily form matrix $D$ from step two of the low rank matrix approximation SVD conversion

$$Q^T A = D$$

multiplying this equation by $Q$ on the left gives

$$QQ^T A = QD,$$

which reduces to the following equation, with dimensions added,

$$A^{m\times n} = Q^{m\times r} D^{r\times n}$$

Notice, this is a partial factorization necessary for the technique described in section II.E.4. Now using the smaller matrix *D*, we compute its SVD giving us

$$D = U_2 \Sigma V^T$$

where $\Sigma$ is a diagonal matrix made up of the nonzero singular values of *D*, which are also the nonzero singular values of *A*, and the matrix *V* is made up of the corresponding right eigenvectors of *D*, which are also the corresponding right eigenvectors of *A*. If the left eigenvectors of *A* are also needed they can easily be computed by the matrix multiplication of $U_2$ on the left by *Q*.

### 6. Businger Golub QR Matrix Factorization with Column Pivoting Algorithm

The most commonly used QR Factorization with Column Pivoting algorithm is the Businger and Golub algorithm (Figure 1) [16], [17]. This algorithm can be broken into two parts with the second part being further divided into 4 subparts and will produce, given a matrix $A^{m \times n}$, the $\Pi$, *Q*, and *R* matrices described in Section 2.E.2.

**Setup:**

Permutation vector: perm($j$) = $j$, $j$ = 1 : n

Column norm vector: colnorms($j$) = $\|A(:, j)\|_2$, $j$ = 1 : $n$

**Reduction Steps:**

For $j$ = 1 : $n$

1. **Pivoting**: Choose $p$ such that colnorms($p$) = max (colnorms($j$ : $n$)

   If (colnorms($p$) == 0) STOP

   If ($j \neq p$) then

   Perm([$j$, $p$]) = perm([$p$, $j$]), A(: , [$j$,$p$]) = A(: , [$p$,$j$])

   Colnorms([$j$,$p$]) = colnorms([$p$,$j$])

   Endif

2. **Reduction**: Determine a Householder matrix $H_j$ such that

   $$H_j A(j:m, j) = \pm \|A(j:m, j)\|_2 e_1$$

3. **Matrix Update**:

   $$A(j:m, j+1:n) = H_j A(j:m, j+1:n)$$

4. **Norm Downdate**:

   $$\text{colnorms}(j+1:n) = \text{colnorms}(j+1:n) - A(j, j+1:n)$$

Endfor

Figure 1. Businger and Golub QR factorization with column pivoting algorithm. From [16]

The permutation matrix, Π, is found by permuting the columns of the identity matrix of the appropriate size according to the permutation vector. $Q$ is found by multiplying the householder matrices, found in Figure 1 step 2, together then subtracting them from the identity matrix and $R$ is what remains in the modified $A$ matrix.

In this chapter, we discussed many of the technologies, algorithms, and previous works that were used in the development of our masterless distributed computation system. In the next chapter, we discuss the tools and the experiments that will be used to show the efficacy of our system.

# III. EXPERIMENTAL SETUP

## A. COMPUTING ENVIRONMENT

### 1. Devices

Two types of devices were used for these experiments: the Samsung Galaxy Tablet and the Motorola Xoom Tablet. The Xoom is running the Google Android 4.0 operating system and the Galaxy is running the Google Android 3.2 operating System. Both devices have a 1 GHz dual-core NVIDIA Tegra 2 processor, are Wi-Fi enabled, and have 1GB RAM. Both devices are limited to 48 MB of heap space per application.

### 2. Network

A Linksys WRT54G router was used as an access point to allow each of the devices to communicate with each other via the 802.11G networking protocol. All of the devices' IP addresses are in the same network, allowing each of the devices to receive all datagrams sent to the network's broadcast IP address. This is a simple way to mimic an ad-hoc network where all devices are within range of all other devices.

### 3. Data Storage

The data used for each of the experiments outlined in III.C and III.D were stored in text files on the SD card of each of the devices. Accessing each of the files is performed by mapping the file into memory which reduces access time to the file and allows for manipulation of the file as if it were stored in memory. This technique was chosen because portable devices generally have lower amounts of RAM, as described in II.B.3, and memory mapping files is a good method to mitigate the performance drop of working with large amounts of data with limited memory. In this case, the devices allow only 48 megabytes of memory per application, which was more memory than was needed for most of the experiments performed, but not for all of them.

## B. MASTERLESS DISTRIBUTED COMPUTATION PROCESS

### 1. Overview

This masterless distributed computing system is comprised of any number of devices all of which have similar processing capabilities, i.e., similar amounts of RAM and CPU speed, and all having exact copies of the distributed computing system running. When the devices are given a task to perform they each begin processing the task. When a portion of the task is embarrassingly parallelizable, this must be recognized by the developer and coded as such, each individual device starts working on the data, even though they are not initially aware the other devices exist, nor do they track the existence of other devices. When the parallelizable portion is completed the devices return to processing the task individually until another parallelizable portion is reached or the process completes.

### 2. Execution

Each device starts each task from the beginning. Nonparallelizable portions of a task are done on each device until a parallelizable portion of a task is reached, at which time cooperation amongst the devices begin. First the parallelizable portion of the task is split into as many independent pieces as possible. Each device chooses a random piece of the split task and begins processing the piece. Note, it is possible that devices will choose the same task to begin with but the likelihood of this happening decreases as the number of tasks increases. When the piece is completed the device can, depending on how the programmer chooses, either broadcast the output of the computation or process more pieces, reducing multiple outputs into a single output, and broadcasting this reduced output. Each broadcast will also specify to which portion of the computation these outputs correspond.

Each device listens for the broadcasts from other devices and when received, these broadcasts are incorporated into the local processing of the task and the pieces designated in the broadcast are marked as completed. Devices process pieces sequentially to make tracking progress and determining redundancies less computationally expensive. When it is determined that an overlap will occur, the device

processes pieces up to the piece that has already been processed by another machine. At this point, since an overlap has been detected, all unbroadcasted answers are reduced and broadcasted out and a new starting position is randomly chosen. When a device determines that all pieces of its parallelizable task have been processed it immediately begins its next task. If that task is not parallelizable then it is processed normally, otherwise the parallelization process begins again. This process is outlined in Figure 2.

1. The program is running on each device and knows how to split the input into *M* pieces.
2. The program running on each device knows that there are *M* tasks and picks a random starting position, *S*, between 0 and *M*-1. That program will then compute task *S*.
3. When the program is finished with task *S* it will store the answer in memory, update its list of completed tasks, and will immediately start computing task *S*+1 mod *M*.
4. Periodically, the stored answers will be sent via UDP to the network's broadcast IP address. If possible the stored answers will be reduced into a smaller answer. All devices will receive this set of answers and will both store the answer locally and update their list of completed tasks.
5. When task *S*+*n* is reached and it is already marked as completed, the program will randomly choose a new *S* from the list of uncompleted tasks and the process repeats from step 2 above.
6. The overall task is complete when all *M* tasks are completed.

Figure 2. Breakdown of steps involved in processing a parallelizable portion of a task.

### 3. Handling Failures

Unlike in the MapReduce system, the failure of nodes is not tested for and will not affect the outcome of the overall parallelizable portion of the task. Because each

device is running its own copy of the computation and will ensure that all portions of its computations are complete, the failure of a device will just mean that the remaining devices will have to process a portion, inversely proportional to the total number of devices, of the pieces the failed node would have processed.

## C. PRELIMINARY COMPUTATIONS

Several preliminary experiments were performed to show the efficacy of the system in purely distributable computations. The first experiment consisted of the multiplication of matrices. In this experiment, both the size of the matrices and the number of devices used for the computation were varied; this was done in order to show the speedup gained by adding devices and the efficiency gained by performing longer computations. The next experiment was the multiplication of a chain of 10 matrices which is intended to show that computations can be chained together resulting in benefits similar to the previous experiment. Again, in this experiment the size and the number of devices used for the computations were varied.

## D. SINGULAR VALUE DECOMPOSITION COMPUTATION

The decomposition of a matrix into its Singular Value Decomposition (SVD) is performed to show that meaningful applications can be performed using this masterless distributed computing system. The method used to perform this calculation is described in II.E.5, the steps of which are reiterated here with annotations of where the algorithm is distributed.

### 1. Calculation of Y=AΩ

This is the most computationally expensive operation for the entire decomposition and this is the step where distribution of the calculation helps the most. In this step, every device constructs $\Omega$, a randomly generated matrix, in a deterministic way, i.e., generating the matrix using the same seed for the random number generator. The size of $\Omega$ is $n \times (k + p)$, where $n$ is $A$'s smaller dimension, $k$ is $A$'s expected numeric rank, and $p < 10$ is a small oversampling parameter. An approximate value of $k$ should be found in

advance of performing the calculations on a mobile device. Foreknowledge of a matrix's attributes can assist with future manipulations of similar matrices [18], accordingly in this work we assign an approximate value to *k*, the rank of *A*, in advance. Each device already has *A* stored locally, so it is known that *Y* should have $m(k+p)$ entries, each of which is considered a task that must be computed as outlined in Figure 2. This task is outlined in Figure 3.

1. Determine the number of elements that make up the answer matrix, *Y*. This is the larger dimension of *A*, *m*, times the rank of *A*, *k*, plus the oversampling parameter, *p*; $m(k+p)$.
2. Randomly choose a starting element, S, between 0 and $m(k+p)$ and calculate its value by multiplying row $\left\lceil \frac{S}{k+p} \right\rceil + 1$ of A by column $\left(S \bmod (k+p)\right)+1$ of $\Omega$.
3. Increment S and calculate its new value as done in step 2. If $S \geq m(k+p)$ then calculate element $S \bmod \left(m(k+p)\right)$.
4. Repeat step 3 until 50 answers have been accumulated, then broadcast the batch of answers to the other devices. If S+1 has already been calculated then broadcast out the answers that have been calculated thus far in the batch and randomly choose a new S from the pool of uncompleted elements.
5. While calculating and sending answers, continually monitor for answers that are being broadcast from other devices. When answers are received from the other devices mark the elements as complete and transfer the answers into the answer matrix, *Y*.
6. When all elements in the answer matrix have been calculated the task is finished.

Figure 3. Steps needed to distribute the calculation of matrix Y.

### 2. Calculation of the *Q* Portion of the QR Decomposition of *Y*

To calculate *Q* the Businger and Golub QR factorization algorithm was used, Figure 1. There is no need to save *R* or $\Pi$ after the computation of *Q* is complete.

### 3. Calculation of $D=Q^{T}A$

Matrix $D$ is calculated by multiplying $A$ on the left by $Q^T$. This step is completely distributable and is outlined in Figure 4.

1. Determine the number of elements that make up the answer matrix, $D$. This works out to be the smaller dimension of $A$, $n$, times the rank of $A$, $k$, which is the smaller dimension of the Q calculated in Section 3.D.2; $nk$.
2. Randomly choose a starting element, S, between 0 and $nk$ and calculate its value by multiplying row $\left\lceil \frac{k}{S} \right\rceil + 1$ of $Q^T$ by column $\left(S \bmod (nk)\right)+1$ of $A$.
3. Increment S and calculate its new value as done in step 2. If $S \geq nk$ then calculate element $S \bmod nk$.
4. Repeat step 3 until 50 answers have been accumulated, then broadcast the batch of answers to the other devices. If S+1 has already been calculated then broadcast out the answers that have been calculated thus far in the batch and randomly choose a new S from the pool of uncompleted elements.
5. While calculating and sending answers, continually monitor for answers that are being broadcast from other devices. When answers are received from the other devices mark the elements as complete and transfer the answers into the answer matrix, $D$.
6. When all elements in the answer matrix have been calculated the task is finished.

Figure 4. Steps needed to distribute the calculation of matrix $D$.

### 4. Decomposition of $D$ into its SVD

The SVD of $D$ is found using the Efficient Java Matrix Library (EJML). This is only possible if $D$ fits into memory because EJML will not use memory mapped files. The size of $D$ varies depending on the size and rank of $A$, but when the rank, $k$, is less than the smaller dimension of $A$, $n$, then $D$ is guaranteed to be smaller than $A$ increasing its chances of fitting into main memory. In our experiments, $k$ is at most 30% of $n$ so the number of elements in $D$ is much smaller than the number in $A$.

### 5. Calculation of *U*

With the SVD of *D* already calculated to be

$$D = U_2 \Sigma V^T$$

the remaining step is to find *U* in the SVD of *A*. This is found by multiplying $U_2$ on the left by *Q* giving us the final factorization of *A* as

$$A = QU_2 \Sigma V^T = U \Sigma V^T$$

This step is also completely distributable and is outlined in Figure 5.

1. Determine the number of elements that make up the answer matrix, *U*. This works out to be the larger dimension of *A*, *m* times the rank of *A*, *k*, which is the smaller dimension of the *Q* calculated in Section 3.D.2; $nk$.
2. Randomly choose a starting element, S, between 0 and $mk$ and calculate its value by multiplying row $\left\lceil \frac{k}{S} \right\rceil + 1$ of $Q^T$ by column $\left(S \bmod (mk)\right) + 1$ of $U_2$.
3. Increment S and calculate its new value as done in step 2. If $S \geq mk$ then calculate element $S \bmod mk$.
4. Repeat step 3 until 50 answers have been accumulated, then broadcast the batch of answers to the other devices. If S+1 has already been calculated then broadcast out the answers that have been calculated thus far in the batch and randomly choose a new S from the pool of uncompleted elements.
5. While calculating and sending answers, continually monitor for answers that are being broadcast from other devices. When answers are received from the other devices mark the elements as complete and transfer the answers into the answer matrix, *U*.
6. When all elements in the answer matrix have been calculated the task is finished.

Figure 5. Steps needed to distribute the calculation of matrix *U*.

In this chapter, we described the experiments that we have conducted to show the efficacy of our masterless distributed computation system and the tools that we used. In the next chapter, we will provide the results of these experiments.

THIS PAGE INTENTIONALLY LEFT BLANK

# IV. RESULTS AND ANALYSIS

## A. OVERVIEW

Four data points were collected from each run of the experiments; time taken, data received, number of tablets used, and CPU usage. The time taken, along with number of devices in this experiment, was used to calculate the speed up achieved by adding devices as

$$speed_up = \frac{T_1}{T_N},$$

where $T_1$ is the time taken when using one device and $T_N$ is the time taken when using $N$ devices. The data received was compared to the calculated amount of data that should have been received. This allowed for the calculation of the redundancy of the run as

$$redundancy = 100\left(\frac{data_received}{data_calculated} - 1\right).$$

The redundancy is equivalent to the percentage of extra work that was done during the experimental run due to overlapped work. In addition to the redundancy, the received data is also used to calculate the average bit rate used by the network during the run. Finally, the CPU usage of each of the runs remained at 100% for each of the runs.

## B. MATRIX MULTIPLICATION

Experiments were performed on square matrices of size M from 50 to 800 being multiplied by matrices of like size. At each size $M$, each answer matrix consists of $M^2$ elements each of which is the result of $2M$ operations (M multiply operations and M addition operations). The speed up of each of the runs is summarized in Figure 6.

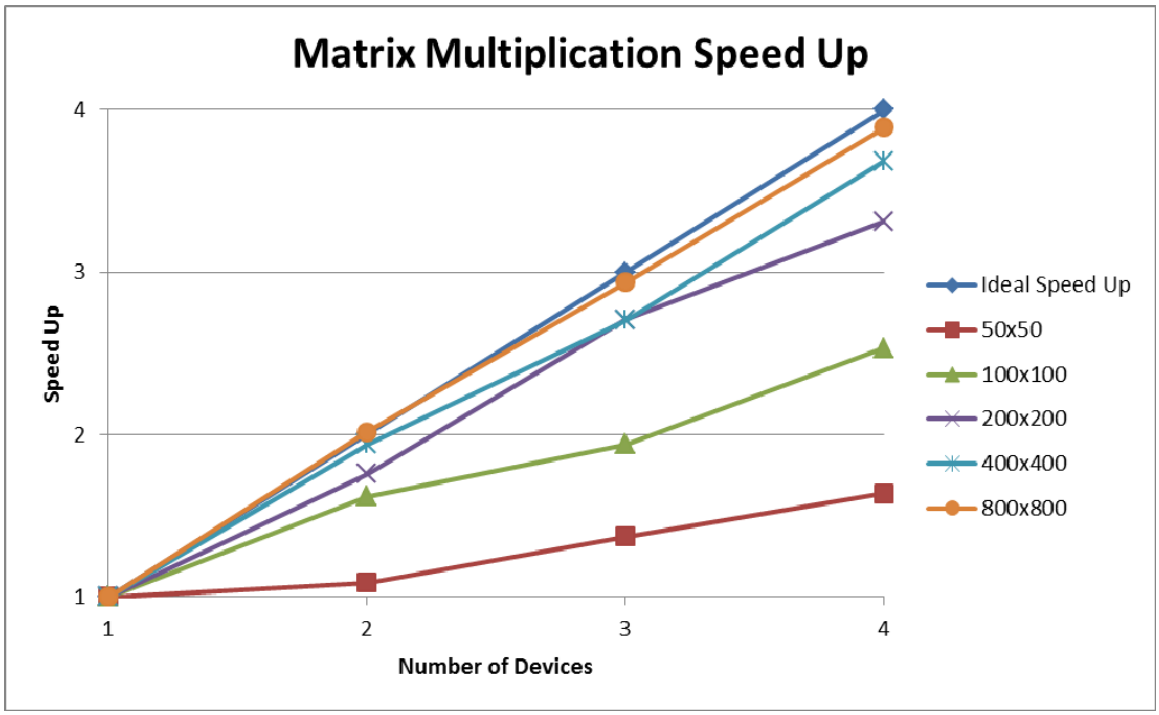


Figure 6. Speed up achieved at each matrix size.

The trend that can be seen emerging in Figure 6 is clear: as matrix size increases the speed up obtained nears the ideal theoretic speed up, proposed by Ahmdal and Gustafson when the time needed to perform the serial portion of the calculation is zero.

There are two possible phenomena that each can explain why multiplication of the larger matrices nears the ideal speed up whereas multiplication of smaller matrices benefits little from the distributed system. First, the time needed to perform the serial portion of the calculation, *s*, is a larger percentage of the overall computation when the matrices are small. This can be seen from Ahmdal's equation

$$T = s + \frac{p}{N},$$

where $p$ is the time needed to perform the parallelizable portion of the calculation and $N$ is the number of devices. The speed up is then calculated as

$$speed_up = \frac{T_1}{T_N} = \frac{s+p}{s+\frac{p}{N}}.$$

In the ideal case, $s$ is zero making the speed up equal to $N$. In the worst case, $p$ is zero making the speed up equal to one, independent of the value of $N$. In the situation that occurs here, $s$ starts closer to $p$ when $p$ is small and becomes proportionally smaller than p as the size of the matrices increases. As $p$ increases the contribution of $s$ becomes negligible, allowing the computations to near the ideal speed up.

The second phenomenon that helps explain the trends of Figure 6 is the amount of duplicated work done during the experimental runs. Figure 7 shows the duplication seen during each of the runs.

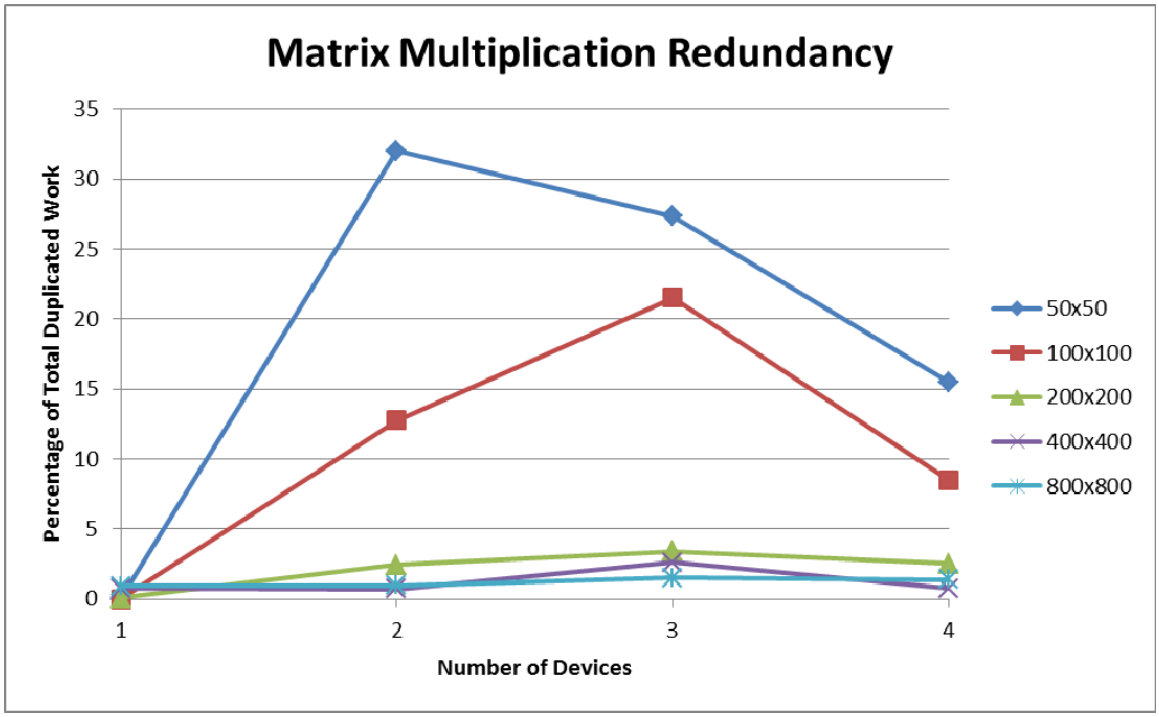


Figure 7. The redundancy observed during the experimental runs.

It can be seen from Figure 7 that there is a much larger percentage of duplicated work done with smaller size matrix multiplications. The source of duplication observed during the experimentation was due to the random starting positions, which are necessary because we did not allow point to point communication between the devices. Unfortunately, this did not explain the reduction in redundancy occasionally observed

when increasing the device number. One possible explanation is that the thread responsible for removing incoming data from the socket queue is underutilized at the beginning of a distributed operation. So, increasing the number of devices possibly reduces this wasted time and allows collisions to be detected earlier. To understand this better one would need to analyze the socket queue usage during a distributed operation.

Starting positions can overlap at any time during a run but the probability of overlap increases as the pool of remaining element's size decreases. The starting size of every answer pool is $M^2$ elements, where $M$ is the number of rows (or columns) of the square matrix, and as elements are processed the pool becomes smaller and smaller thereby increasing the likelihood of duplication. Duplication becomes more and more likely as the pool size decreases, becoming unavoidable when the pool gets too small. This means that the chance of overlap at any time is determined by the elements remaining in the pool to be processed and the number of devices performing the processing.

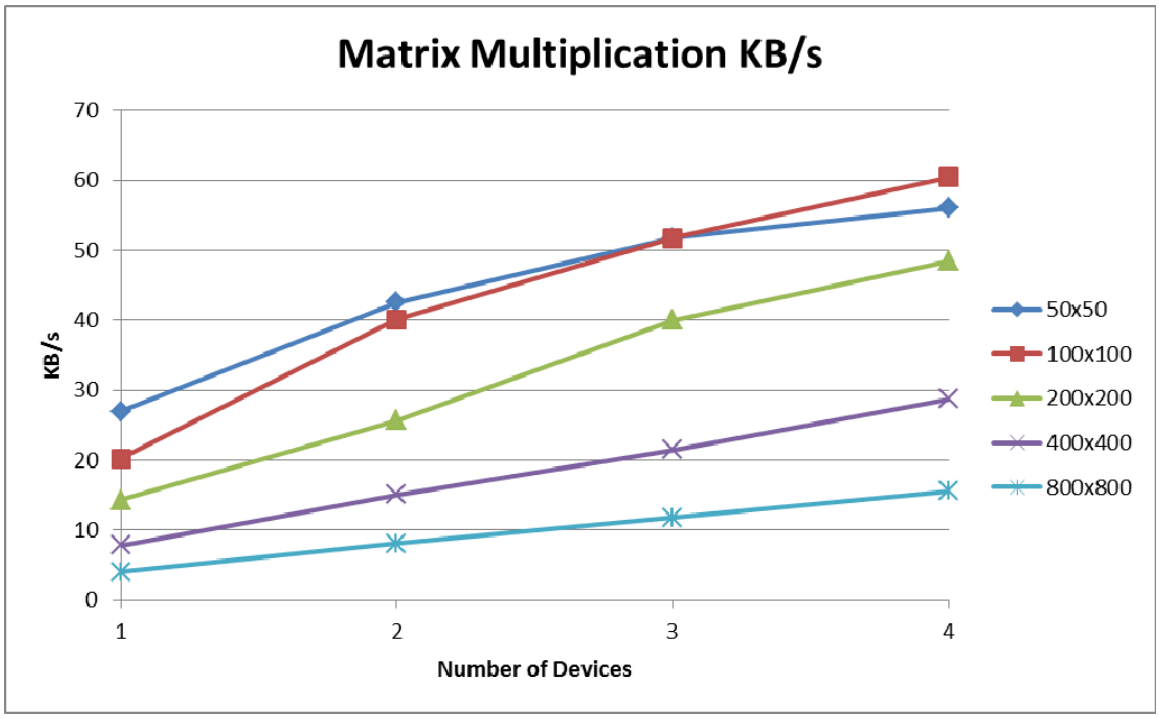


Figure 8. Network usage during matrix multiplication.

The two trends that appear in Figure 8 are that as the number of devices increases so does the KB/s and that as the matrix size increases the network usage decreases. As the updates are received on the devices they are put in a receive buffer, from which they are pulled then integrated into the computation. As was stated in IV.A, the CPU usage is a consistent 100% throughout the entirety of the computation which causes the receive buffer to fill faster than updates can be pulled off of it. When the buffer is full all new updates are discarded until updates are pulled, thereby freeing space.

## C. CHAIN OF TEN MATRIX MULTIPLICATIONS

In Section III.C, we decided this was performed to show that computations could be performed without signaling either the beginning of or end of an operation. The results found show that a speed up similar to the single matrix multiplication could be achieved and are summarized in Figure 9.

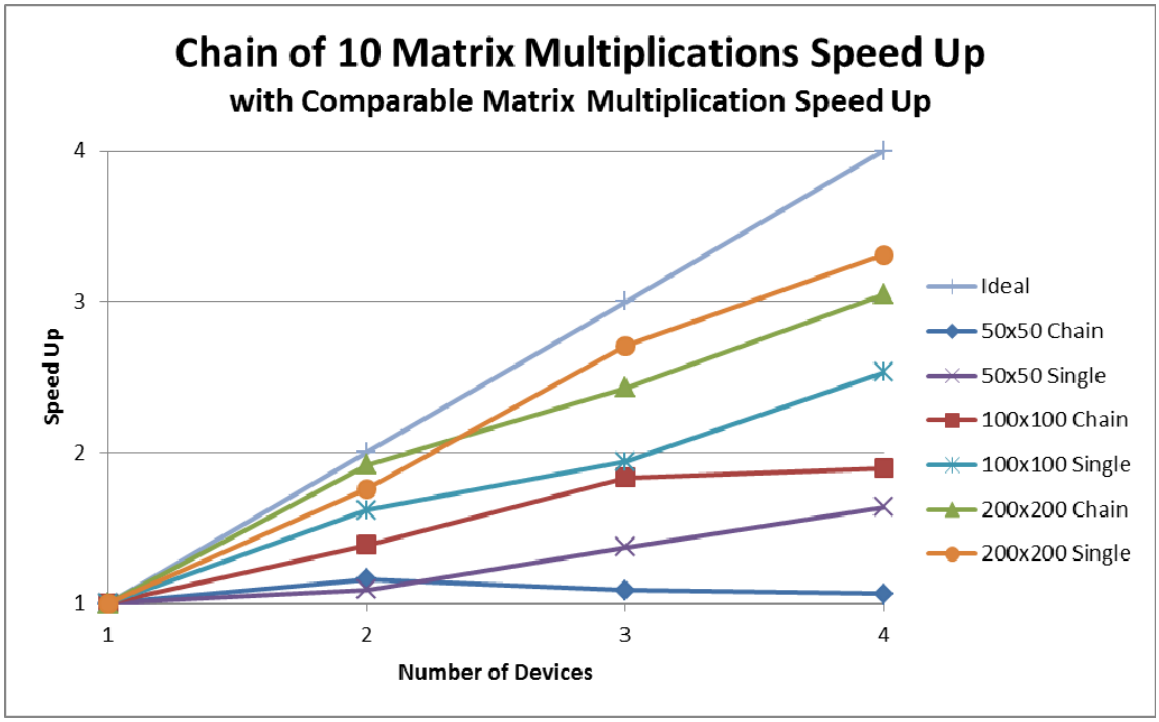


Figure 9. Speed up obtained by performing ten matrix multiplications.

These results differ little from those reported in Chapter IV.B and the same trends are still apparent. It is noticeable though that the speed up obtained in the chain of matrix multiplications is lower than the single matrix multiplication, but not by much. Similar results were found for the redundancy, Figure 10, and the network speed, Figure 11.

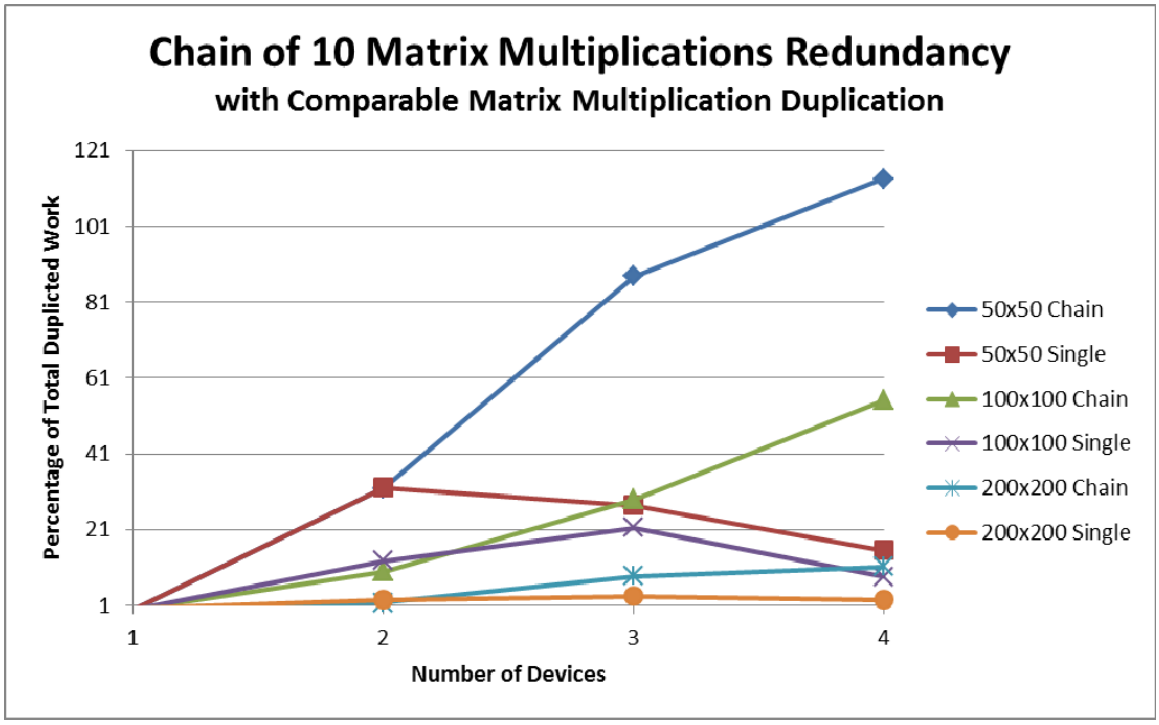


Figure 10. Redundancy observed in Chain of 10 Matrix Multiplications.

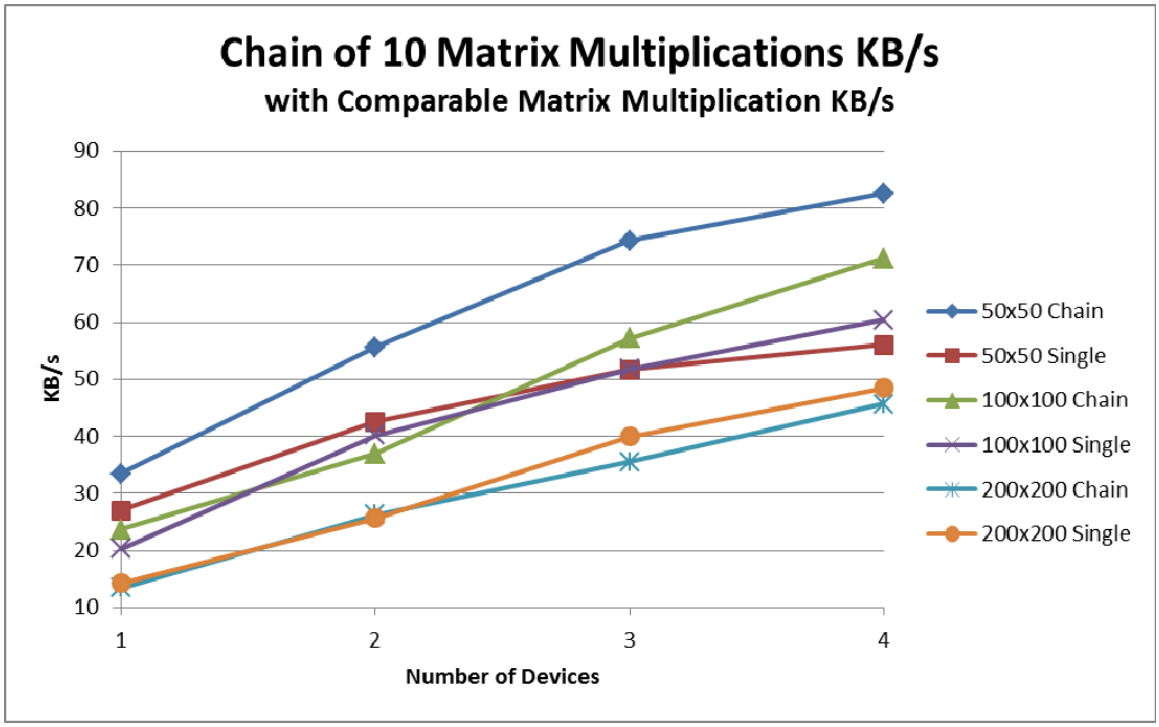


Figure 11. Network usage during chain of multiplications.

It was expected that the chain of multiplications would be worse with respect to speed up and redundancy but it was not expected that the network usage would noticeably change. The increase of redundancy was expected because it was thought that the receive buffers would remain full at the end of an operation and would have to be emptied before new data could be received. During the time that the buffers would be full with updates from previous multiplications they would be dropping the needed updates from the current operation. So, whereas a single computation would start with an empty buffer, nine of the ten multiplications would begin with full buffers. It is this increase of redundancy that led us to believe that speed up would suffer accordingly.

It is also noticeable that as the matrix size increases each of the comparable trends tend to converge. This is particularly noticeable in the network usage where the two 200x200 trends are nearly equal. This overall trend can be seen in Figure 12.

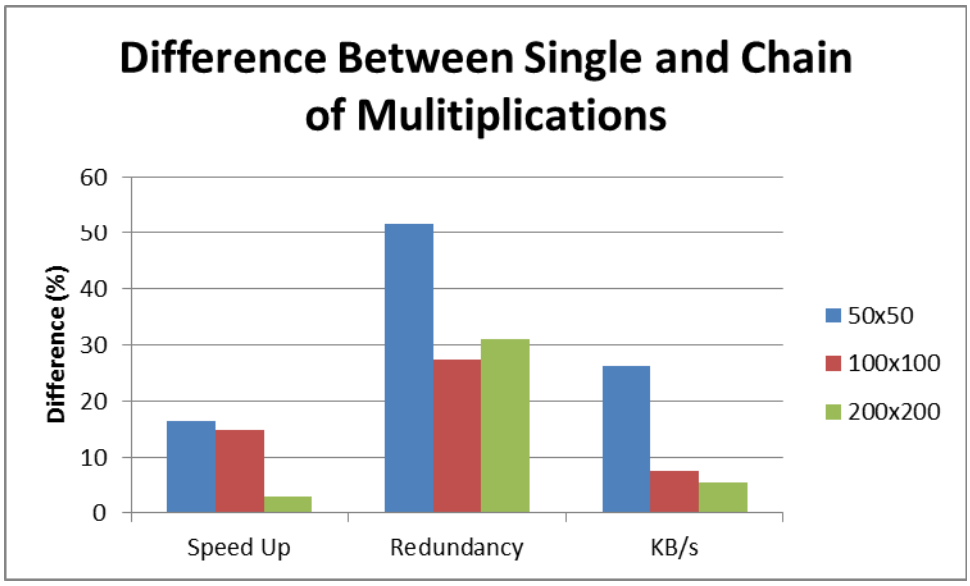


Figure 12. Difference between single and chain of multiplications. The percentage difference between the values measured during the single matrix multiplication and the chain of multiplications is clearly decreasing as the matrix size increases.

## D. DISTRIBUTED SVD

When performing the SVD over this distributed system the size of the matrix, *A*, was varied between 500 and 1500 and its rank was varied between 50 and 150. This experiment shows that both complex linear algebra operations can benefit from this system and that this system is effective even when large serial operations are performed between the parallelizable operations. The speed up obtained performing the SVD on matrices of size 500x500 are shown in Figure 13.

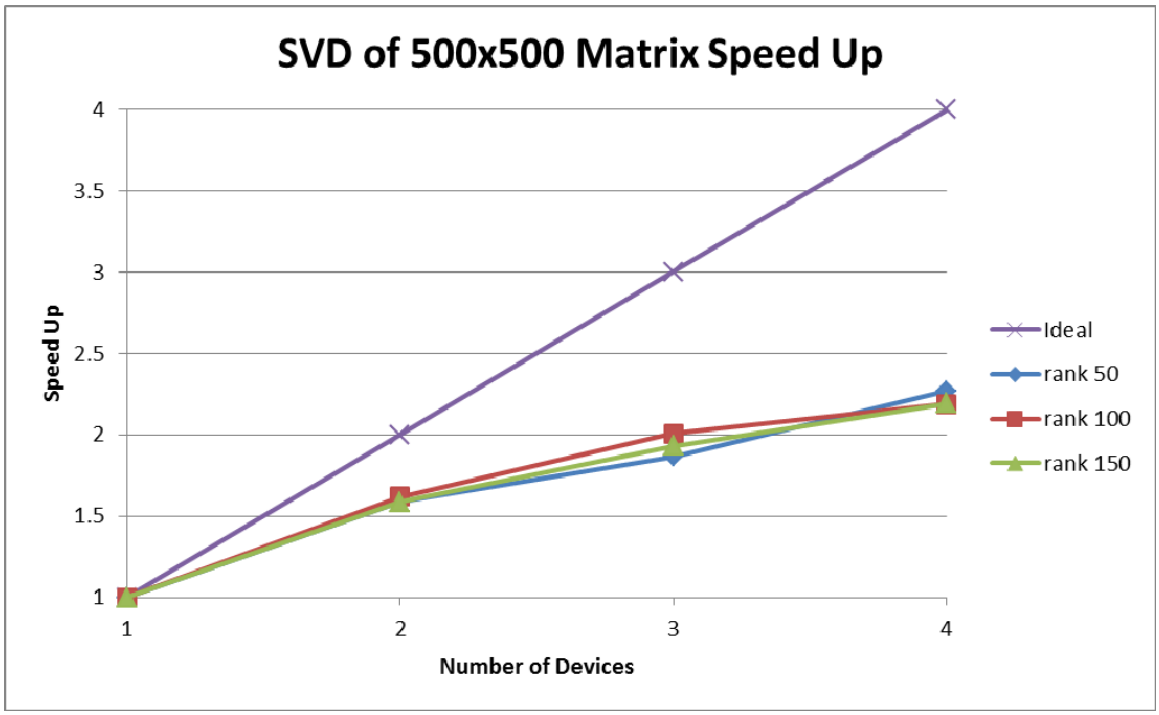


Figure 13. Speed up of the SVD obtained on the 500x500 matrix.

The range of the ranks chosen has very little affect on the speedup obtained when computing the SVD of a 500x500 matrix. Similarly, when computing the SVD of a 1000x1000 and 1500x1500 it is shown that the range of ranks chosen has little affect on the overall speed up. But, as the matrix size increases, it can be seen that the speed up does improve as shown in Figure 14.

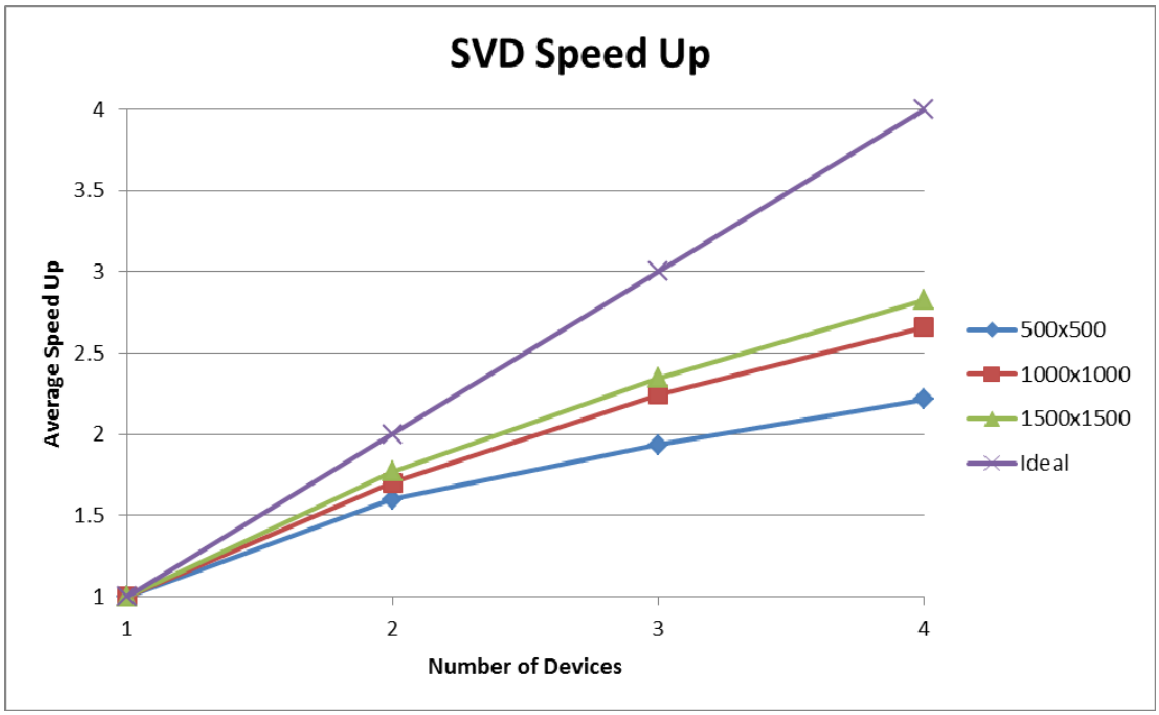


Figure 14. Average speed up of varying size SVD computations.

It is assumed that the rank of the matrices has little effect on the speed up of the SVD computations only for such small values. As the rank approaches *N*, the smaller dimension of the matrix *A*, the speed up goes to 1 because the computation becomes more and more unparallelizable. This was not able to be shown using these devices due to the limited amount of RAM available. In addition, it is recognized that if the rank of a matrix is near *N* then performing the extra steps of determining the rank of A actually increases the time that would be needed to compute *A*'s SVD [12]. This point is moot for this experiment because the mobile devices were unable to compute the SVD of a large matrix thus necessitating the need to determine *A*'s rank.

In this chapter, we showed that masterless distributed computing is feasible for computations of various sizes and as the size of the computation increases, the speed up obtained by adding devices nears the theoretic possible speed up. In addition, through the decomposition of a matrix we showed that these operations can be chained together in useful ways. In the next chapter, we will conclude our discussion on this masterless computing system and provide ways that this research can be continued.

# V. CONCLUSION AND FUTURE WORK

## A. SUMMARY

There is a need to analyze large amounts of data in hostile environments that might not have a robust network infrastructure and in this work we described a system that can perform distributed computations in such an environment. The previous works that we used to create our system were presented along with the works used to develop our method to test the system. Normally, the singular value decomposition is not easily distributable, especially without tight synchronization. In this work, we described and used a method to decrease the complexity of the decomposition while minimizing the resulting error in the decomposition. We also showed the results of our various experiments that illustrated the efficacy of our system.

## B. FUTURE WORK

### 1. Implement Masterless Distributed Computing System in an Ad-hoc Networking Environment

Our system was designed to use broadcasts to overcome a lack of infrastructure. This was possible because our system was masterless, whereas if our system did contain a master than devices would require an infrastructure to ensure efficient communication between a master and the workers. Although our system was designed with these deficiencies in mind, we still implemented the system using a wireless accesspoint in order to make sure that there was network connectivity. This system should be reimplemented in an ad-hoc network environment and reanalyzed to ensure its efficacy in such an environment.

### 2. Analyze the Socket Layer Receive Buffers

In Section IV.B, we described an unexpected result where increasing the number of devices actually reduced the amount of duplicated work. Our original intuition was that increasing devices would increase the amount of work duplicated but this was not always the case. One possible explanation is that the thread responsible for removing

data from the socket layer receive buffers was being underutilized. An analysis of the buffer during a distributed computation should be done to better understand this phenomenon.

### 3. Test Efficacy of the System when Devices are Both Entering the System and Leaving the System

We designed our system to continue working in the event of a device, or multiple devices, being removed from the system. Although this ability is inherent in our system we did not test the affect a device leaving will have on completion times or network usage. In addition, a "catch-up" mechanism can be implemented to allow devices entering the system to aide in the distributed computation. In addition, the affect a device entering the system has on the overall system should be analyzed.

### 4. Analyze the System's Abilities to Work with Other Types of Computations

This system was designed with Google's MapReduce in mind and should work with most algorithms implemented in MapReduce. One large difference is that each individual device must be able to *reduce* the results of its *mapping* stage so as to efficiently broadcast to the other devices. This decreases the amount of MapReduce algorithms that will work in this system but there are still many that should work. Additionally, analysis may show that some algorithms work particularly well in this system; for instance algorithms with extensive *mapping* phases that *reduce* to binary answers.

## C. CONCLUSION

Our research goal was to see if we could construct a masterless distributed computing system that does not rely on network routing, and if we could determine whether the system could be used for solving computationally expensive linear algebra problems. Our results showed that our system does indeed speed up a distributed calculation and does it in a way that does not rely on the presence of a routable network. We also showed that the speedup obtained nears optimal as the size of the computation necessary to calculate an update increases. Additionally, we have shown that we can

chain distributed computations together resulting in a decreased amount of time needed to perform a useful calculation, the singular value decomposition of a matrix.

The implications of these results is that masterless distributed computing in an infrastructureless environment is feasible. This research may one day aid in a commanders ability to analyze battlefield conditions and develop optimal strategies to accomplish their mission.

THIS PAGE INTENTIONALLY LEFT BLANK

# LIST OF REFERENCES


[1] P. Marshall, "DARPA progress towards affordable, dense, and content focused tactical edge networks," *MILCOM 2008 - 2008 IEEE Military Communications Conference*, pp. 1–7, Nov. 2008.

[2] N. Suri and G. Benincasa, "Peer-to-peer communications for tactical environments: Observations, requirements, and experiences," *IEEE CommunicationsMagazine*, October, pp. 60–69, 2010.

[3] P. Ewing, "Army begins mobile phone experiments," *dodbuzz.com*, 2011. [Online]. Available: http://www.dodbuzz.com/2011/06/06/army-begins-mobile-phone-experiments/. [Accessed: 15-Feb-2012].

[4] G. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18–20, 1967, Spring Joint Computer Conference*, 1967, vol. 30, pp. 483–485.

[5] J. Gustafson, "Reevaluating Amdahl's law," *Communications of the ACM*, vol. 31, no. 5, pp. 532–533, 1988.

[6] Google, "Android - Discover Android," *Google Developers*, 2012. [Online]. Available: http://developer.android.com/guide/basics/what-is-android.html.

[7] D Bornstein, "Dalvik VM internals," 2008 Google I/O Session Videos and Slides. [Online]. https://sites.google.com/site/io/dalvik-vm-internals

[8] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[9] C. Shi, V. Lakafosis, and M. Ammar, "Serendipity: A distributed computing platform for disruption tolerant networks," M.S. thesis, Georgia Tech, Atlanta, Georgia, 2011.

[10] L.-L. Xie and P. R. Kumar, "A network information theory for wireless communication: Scaling laws and optimal operation," *IEEE Transactions on Information Theory*, vol. 50, no. 5, pp. 748–767, May 2004.

[11] J. Li, C. Blake, D. S. J. De Couto, H. I. Lee, and R. Morris, "Capacity of Ad Hoc wireless networks," in *Proceedings of the 7th annual international conference on Mobile computing and networking - MobiCom '01*, 2001, pp. 61–69.

[12] N. Halko, "Randomized methods for computing low-rank approximations of matrices," Ph.D. dissertation, University of Colorado, Boulder, Colorado, 2012.

[13] P. R. Peres-Neto, D. a. Jackson, and K. M. Somers, "How many principal components? stopping rules for determining the number of non-trivial axes revisited," *Computational Statistics & Data Analysis*, vol. 49, no. 4, pp. 974–997, Jun. 2005.

[14] N. Halko, P. G. Martinsson, and J. A. Tropp, "Finding structure with randomness: probabilistic algorithms for constructing approximate matrix decompositions," *SIAM Review*, Vol. 53, No.2, pp. 217–288.

[15] Y. Hong, "Rank-revealing QR factorizations and the singular value decomposition," *Mathematics of Computation*, vol. 58, no. 197, pp. 213–232, 1992.

[16] G. Quintana-orti, X. Sun, and C. Bischof, "A BLAS-3 Version of the QR Factorization with Column Pivoting," *SIAM Journal on Scientific Computing*, vol 19, issue 5, pp. 1486–1494, 1998.

[17] C. Loan, "A Survey of Matrix Computations," *Handbooks in OR & MS*, vol. 3, pp. 247–321, 1990.

[18] R. Barrett et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1994.

# INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California